Demystifying Power and Performance Bottlenecks in Autonomous Driving Systems

Pedro H. E. Becker, José María Arnau, Antonio González
Department of Computer Architecture
Universitat Politècnica de Catalunya
{pedro, jarnau, antonio}@ac.upc.edu

***Abstract*—Autonomous Vehicles (AVs) have the potential to radically change the automotive industry. However, computing solutions for AVs have to meet severe performance and power constraints to guarantee a safe driving experience. Current solutions either exhibit high cost and power dissipation or fail to meet the stringent latency constraints. Therefore, the popularization of AVs requires a low-cost yet effective computing system. Understanding the sources of latency and energy consumption is key in order to improve autonomous driving systems. In this paper, we present a detailed characterization of Autoware, a modern self-driving car system. We analyze the performance and power of the different components and leverage hardware counters to identify the main bottlenecks.**

Our approach to AV characterization avoids pitfalls of previous works: profiling individual components in isolation and neglecting LiDAR-related components. We base our characterization on a rigorous methodology that considers the entire software stack. Profiling the end-to-end system accounts for interference and contention among different components that run in parallel, also including memory transfers to communicate data. We show that all these factors have a high impact on latency and cannot be measured by profiling isolated modules.

Our characterization provides novel insights, some of the interesting findings are the following. First, contention among different modules drastically impacts latency and performance predictability. Second, LiDAR-related components are important contributors to the latency of the system. Finally, a modern platform with a high-end CPU and GPU cannot achieve real-time performance when considering the entire end-to-end system.

***Index Terms*—autonomous vehicles, workload characterization, performance analysis**

I. Introduction

Autonomous Vehicles (AVs) are a promising solution for a better organized, more inclusive, and safer traffic [1], [2]. These appealing benefits had put AVs on the track of many leading tech companies. Some prominent industry projects include Waymo's AVs [3] (formerly Google's self-driving car project), the NVIDIA's Drive Platform [4], Tesla's autopilot [5], and Baidu's Apollo project [6]. Despite industry progress, AVs are not yet on the large-scale production and still a case of research. To reach market-level maturity, AVs must be first-rate regarding safety, performance, and power-efficiency.

AVs rely on a complex chain of algorithms to perform the perception of the surrounding world (e.g., object detection and object tracking) that is used to perform the driving decisions (e.g., steering, accelerating, or braking). For safety reasons, the computation tasks are constrained by time deadlines, assuring the vehicle's reaction to the traffic events is in time to avoid accidents. Not surprisingly, the computing power of AVs is under pressure for improvement. In a recent report [7], Huawei states that the future's full-sell driving technology should be able to deliver 100× the performance (in TOPS) of nowadays driving assistance solutions. Furthermore, since AVs rely on power-hungry hardware, usually a combination of high-end CPUs and GPUs, power and energy consumption must also be taken into account. If energy consumption is too high, the autonomy of the vehicle will be reduced [8]. Also, the power dissipation from the computing platform adds extra cooling necessities, further increasing energy consumption.

To overcome these challenges and assure the improvement of AVs, it is mandatory to have a broad yet deep understanding of their software stack and its interaction with underlying hardware, investigating bottlenecks and issues in these complex architectures. Unfortunately, few previous works had put effort into identifying these bottlenecks and quantifying them in actual numbers. Moreover, industry leaders generally restrict the knowledge of their projects within their company, because of market competition, making it hard for outsiders to have a grasp on the open problems for state-of-the-art AVs.

In this work, we present a thorough characterization of a self-driving architecture, detailing open problems in current software and hardware for future research on AVs. The investigation is performed with a modern and fully open-sourced solution, namely Autoware [9], [10], which is built upon cutting-edge algorithms for AVs. We stimulate Autoware's software stack with real-life sensor data and profile several of its traits. The measurement includes the individual computation latency of different modules, the end-to-end latency from sensor data collection up to complete scene recognition and the power and energy required for those tasks.

Some of the novel findings are the following:

- Depending on the image detection method chosen, tail latency of other components varies between 34% and 97% due to contention. Profiling nodes in isolation leads to a significant underestimation of latency and predictability.
- Autoware cannot guarantee real-time perception on a modern computer with a high-end GPU, as its end-to-end latency frequently exceeds time requirements by more than twofold.
- LiDAR-related components, that are key to drive the car safely, are important contributors to end-to-end latency,

showing execution times in the order of tens of ms.

II. Autonomous Vehicles Architecture Overview

To introduce the main concepts of AVs we leverage the Autoware project [9] as a representative cutting-edge solution. Autoware was first introduced in 2015 [11], and has been constantly improved by the open-source community, including significant contributions from industry companies [10]. Although Autoware is not the only open-source project available, it has the largest autonomous driving community on GitHub and does not impose limitations as other projects in the same tier of maturity and complexity. For instance, Baidu's Apollo project, which is also open-source, has some vital parts of the code, such as Deep Neural Network (DNN) models for visual detection, released as black-box libraries [12].

In a high-level, AVs technology can be divided into a set of layers that interact and collaborate to perform self-driving, as we depict in Figure 1. The next subsections detail the modules that compose each of these layers.

A. External Data

We start explaining the external data that feeds the AV computing system, such as sensors data and HD maps.

Sensing. A mandatory layer in AV platforms is composed by the vehicle sensors, whose data will be further processed to characterize the driving scene. Although different companies use different sets of sensors, a list of common sensors includes i) one or more *cameras*, which capture images for object detection; ii) *Light Imaging Detection and Ranging (LiDAR)*, that measures the distance from the car to surrounding objects through laser scan. LiDAR generates a collection of points (named point-cloud) which is used for fine-grain localization and object detection; iii) *RADAR*, for object detection in higher distance ranges compared to LiDAR, but with lower precision; iv) *Global Navigation Satellite System (GNSS)*, to provide an approximate position to the localization algorithms; v) *Inertial Measurement Unit (IMU)*, which provides data such as linear velocity, useful to refine localization.

Autoware interfaces with all previously mentioned sensors except RADAR (as in Figure 1), which is under development. It is also worth to mention that LiDAR is the only sensor among Autoware's minimum requirements [13]. This is due to the versatility of LiDAR generated point-clouds, that can be used for both localization and object detection.

HD Map. AVs require very high precision when localizing themselves [14]. For this reason, commodity map applications (e.g. Google Maps) do not apply for AVs. Instead, localization algorithms generally depend on pre-existent High-Definition (HD) Maps. These HD maps contain a detailed point-cloud of the whole drivable area, which is used by localization algorithms to reach centimeter-level precision, orders of magnitude more accurate than GNSS's meter-level precision. Building an HD map requires precise calibration and is often performed by specialized companies. After collecting the point-cloud of the mapped area, the map is enriched with useful information such as disposition of lanes, allowed ways, speed limits, and

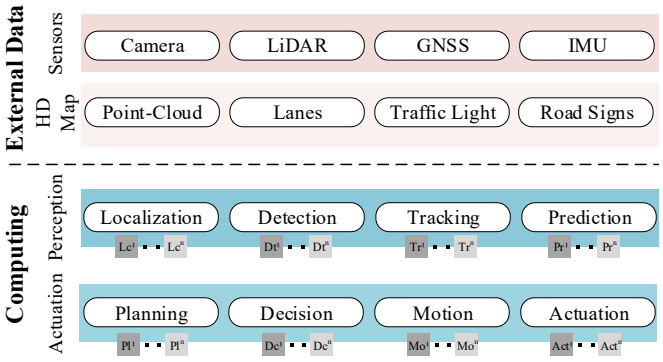


Fig. 1: A high-level description of the Autoware architecture. Sensor data and pre-existing high-definition maps feed the software stack.

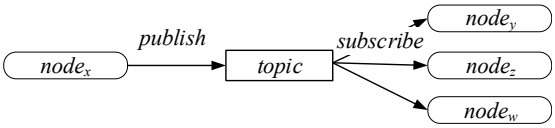


Fig. 2: The publisher-subscriber scheme for *node* communication in ROS. A *node* publishes its output into a memory space, called *topic*. Multiple *nodes* can subscribe to this *topic*, being notified by ROS when new messages are published.

3D position of traffic lights, road signs, and zebra crossings. All this information can be used by the AV system to improve its surroundings perception and actuation.

B. Computing

Autoware software stack is built upon the Robot Operating System (ROS) [15]. ROS is a middleware collection of libraries, tools, and underlying infrastructure that increases productivity on robot systems development. The project is divided into multiple software modules, named *nodes*, whose purpose is to individually solve a task (e.g., detect an object) while globally collaborating towards a major goal (e.g., successfully self-drive a vehicle). The *nodes* communicate among each other through a publish-subscribe arrangement: *nodes* can publish their outputs into a shared memory space, referred as a *topic* in ROS jargon, which other *nodes* can subscribe to, as we depict in Figure 2. When new messages are published, all *subscriber nodes* are notified, being able to read and process data. This simple solution allows *node* collaboration.

Additionally, ROS *topics* specify an interface for message publication, thus supporting different *nodes* implementations as long as the interface is respected. For example, Autoware has multiple *nodes* that perform object detection. Users can quickly choose which image detector implementation they want during a given autonomous driving execution.

Following, we review the most relevant *nodes* from the Autoware software stack.

Perception. Prior to performing driving actions, self-driving vehicles must first understand the scene they are into, through

perception algorithms. As we depict in Figure 1, the perception layer is undertaken by different modules:

- *Localization* is one of the major tasks to be performed by an AV. In Autoware, the process of localization starts with down-sampling the input point-cloud, acquired from LiDAR sensor, through a node called *voxel_grid_filter*[1]. This filtered point-cloud is then processed by the *ndt_matching* [16] node, which tries to match the sensed point-cloud with a sub-part of the pre-existing HD map point-cloud. At this point, the GNSS can indicate an initial position for the matching algorithm to start its search, speeding up the localization. Additionally, the IMU may be used to anticipate where the subsequent positions are likely to be. When the algorithm finishes, the best matching between the LiDAR's acquired point-cloud and the map is selected as the current position for the car with centimeter-level precision.
- *Detection* must identify objects in the environment, classify them (e.g., a car, a pedestrian), and assert their relative localization. In Autoware, object detection can be performed with both LiDAR and *camera* data. We consider the *euclidean_cluster* node for LiDAR detection, which clusters points nearby each other to perceive them as objects, although it cannot classify their type. The algorithm also calculates the cluster centroids to stipulate how distant the objects are from the vehicle. For image-based object detection, Autoware provides support for the latest versions of SSD [17] and YOLO [18]. After detecting and classifying objects in the images, the output is fused with the LiDAR-based detection. This task is completed by the *range_vision_fusion node*, based on calibration between camera and LiDAR devices. This fusion provides several benefits to detection. On the one hand, LiDAR detection adds a 3D perspective to the image-based detection (giving a sense of volume to the object), and also localizing them in the map. On the other hand, image detection adds semantic to the objects, through the classification (e.g., vehicle, pedestrian...).
- *Tracking* aims at keeping an identification of different objects along with successive frames, in order to determine how the different traffic participants are moving. This can be performed based on detected objects (from *Detection* task) and the *ray_ground_filter* node, which filters the point-cloud acquired by LiDAR to separate the ground points and the non-ground points (above ground level). The *tracking* must cope with several challenges since detected objects may experience occlusion, dissimilar motion patterns and clutter [19]. To cope with these problems, Autoware has the *imm_ukf_pda_tracker* node, which is inspired in previous works [20], [21] that combine different filter algorithms to overcome the challenges. The *Tracking* then publishes the list of tracked objects with information such as position, velocity, and associated identification.
- *Prediction* takes place after *tracking* updates the status of the objects. This module utilizes the object's current position, velocity, and direction to stipulate a path they are likely to follow in the future. Currently, Autoware considers the objects have constant velocity (both when driving straight as when turning), hence the prediction *node* name *naive_motion_predict*. Finally, the tracked objects associated predicted paths, and the LiDAR point-cloud feed the *costmap_generator node*. This *node* generates the available areas for the vehicle to drive (i.e., not occupied by objects or to be occupied in the near future, based on the trajectory predictions). This is key for finding the possible trajectories the AV itself can take.

Actuation. Once the vehicle has a broad comprehension of the current traffic scenario and its participants, it can plan and execute driving actions through the following components:

- *Planning* determines the trajectory for the vehicle. In Autoware, it is divided in two parts: global and local planning [22], respectively implemented by *op_global_planner* and *op_local_planner* nodes. The global planner defines a high-level route to reach the target destination. The local planner details how the route will be followed depending on the perception outcome.
- *Motion* modules generate the control output to maneuver the vehicle in order to follow the planned path. Autoware implements the pure pursuit algorithm [23] (*pure_pursuit node*) to calculate the linear and angular velocity the vehicle should perform. These velocity values are submitted to a low-pass filter (*twist_filter node*), used to smooth the driving actions. Finally, Autoware interfaces with the vehicle through a drive-by-wire system, sending the control performing the driving actuation.

III. Methodology

In this section, we present the methodology for the Autoware characterization. We detail tools, configurations, and steps to acquire the data.

A. Autoware Execution Environment

Prior to characterize Autoware, we need to prepare its execution environment. Setting up Autoware in a vehicle is costly and inhibit the reproducibility of the experiments. Instead, we rely on a set of ROS tools and data collected during a vehicle drive to provide trustworthy data input to the Autoware software-stack. Figure 3 introduces how we arrange the experimentation. We start by stimulating Autoware with two inputs: sensor data (in the form of a *ROSBAG* file), and a *point-cloud map*.

The sensor data (*ROSBAG* file) contains ROS topics previously collected during a real-world vehicle driving. This means that we can use sensor data as an input as if Autoware was

[1]There are other down-samplers available. We do not try out an exhaustive combination of all possible implementations, but rather, use the recommended algorithms accordingly with Autoware documentation [10]. This also holds for other tasks, whenever there is a consensus on a given solution. When the best algorithm/implementation is not clear or incurs considerable trade-offs (e.g. object detection DNNs) we experiment the different possibilities.

TABLE I: Summary of important Autoware nodes.

Node	Description
voxel_grid_filter	Downsample an input point-cloud, reducing the amount of points to simplify further computations.
ndt_matching	Localize the vehicle by matching LiDAR acquired point-cloud with a region of the HD map point-cloud.
euclidean_cluster	Cluster LiDAR acquired points nearby each other, identifying volumes that can be perceived as objects.
YOLO / SSD	DNN-based nodes used to detect and classify objects (e.g., vehicles, pedestrians) from images.
range_vision_fusion	Combine LiDAR and image-based detected objects into the same coordinates, improving objects perception.
ray_ground_filter	Separate an input point-cloud in two: points that compose the ground, and points above the ground level.
imm_ukf_pda_tracker	Track objects by assigning them an identification and keeping it coherent among subsequent frames.
naive_motion_prediction	Extrapolate the current trajectory of different objects to predict where they will be in the future.
costmap_generator	Determine drivable areas in the map, i.e., with no objects at the time or predicted to be in the near feature.
op_planner	Global and local path planning based on the current scene and target location.
pure_pursuit	Calculate the necessary motion (linear and angular acceleration and velocity) to follow the desired path.
twist_filter	A low-pass filter applied over motion control to smooth the vehicle driving.

receiving the data from the sensors from a vehicle in real-time. More importantly, we can re-run the same *ROSBAG*, to stimulate Autoware with the same data from sensors as many times as needed. Hence, we can experiment with Autoware with multiple combinations of nodes, metrics, and profiling tools, always using the same input, improving reproducibility, and easing further data analysis. In our experiments, we rely on data from an 8 minutes driving, recorded in the city of Nagoya in Japan, which is available at the *Autoware Data repository* [24].

To completely stimulate the Autoware stack we also need an HD map, as explained in Section II-A. However, HD maps are often proprietary and cover very small parts of the globe. Moreover, the HD map should cover the same region where the original *ROSBAG* file was recorded, but the sensor data we use is not accompanied by an HD map. To overcome this, we use an Autoware utility (*ndt_mapping*), which creates a point-cloud map based on the LiDAR data from the *ROSBAG* file. This, however, only generates a point-cloud map, useful to stimulate the localization nodes, but does not contain the whole HD map with annotations (speed limit, traffic light poles, etc.). We discuss the limitations of this process later on.

The specific hardware and software upon which nodes run on are detailed in Table II. We highlight most of the software versions presented in the Table are required by Autoware, for compatibility reasons. As for the different knobs to configure the nodes, we use the predetermined configuration released with the Autoware (Autoware.ai 1.12 in our case). The vision detection models and pre-trained weights were also obtained following the Autoware guidelines. The SSD vision detector

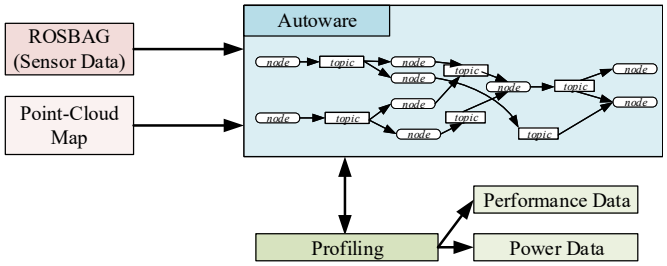


Fig. 3: The experimental setup. Preexistent files for sensor data and point-cloud map provide the real-life inputs. Profiling tools collect characterization data.

TABLE II: Environment hardware and software platforms

		CPU	GPU
Hardware	Model	Intel i7-7700K	NVIDIA GeForce GTX 1080
	Architecture	Kaby Lake	Pascal
	Frequency	4.2 GHz	1.6 GHz
	# Cores	4 (8 Threads)	2560
	L1 Cache (I/D)	32 KB (8-Way)	-
	L2 Cache (Unified)	256 KB (4-Way)	4 MB
	L3 Cache (Unified)	8 MB (16-Way)	-
	Main Memory	64 GB DDR4	8 GB GDDR5X
Software	Operating System	Ubuntu 16.04 x86_64	
	Kernel	4.15.0-96-generic	
	ROS	Kinetic	
	Autoware	Autoware.ai 1.12	
	CUDA	9.0	

is assessed with models SSD300 and SSD512, with data from [25]. The YOLO vision detector was assessed in its third version, namely the YOLOv3-416 model and pre-trained weights from [26].

B. Characterization Procedures

We divide the characterization into three main steps: i) latency measurement, ii) system-wide utilization and power dissipation, and iii) architecture-level characterization. Following we detail the purpose of each of these steps and how they were performed.

Latency Measurement. AVs are subject to time constraints, since assuring the vehicle always finish the computation within a given time limit is necessary to guarantee safety standards. Some previous works consider a reaction time under 100ms [8], [27], in order to be faster than a high-skilled human reaction. This time constraint poses the AV to be four times faster than the average time humans take to process objects, as studied in [28]. Naturally, reducing the computation latency even further is also beneficial since it improves reaction time achieving higher safety standards. For this reason, it is vital to assess the AV's computation latency on Autoware.

We leverage code instrumentation to measure two latency values: *single node latency*, and perception *end-to-end latency*, as illustrated in Figure 4. *Single node latency* measures the

amount of time to process the input, and it is defined as the elapsed time from the moment an input arrives at the node until the output is ready. This will serve to identify latency-hungry nodes, which are potential bottlenecks. We augment the code with C++ *chrono* library calls to collect the execution times along the whole driving.

We also assess *end-to-end latency* for perception. For this, we first define different computation paths, as pointed out in Figure 4. Each computation path corresponds to the accumulated time a given sensor input entered the system until its final contribution to the vehicle's perception was performed. In this case, the input is processed along with different nodes, and thus the computation path latency will include both the computation time inside each node and also the communication time (ROS subscribe-publish). The *end-to-end latency* is defined as the computation path that takes the longest time to finish. Figure 4 depicts this situation, where inputs take different execution paths along with the nodes of the system, and the worst-case computation path equates to the perception's *end-to-end latency*.

Differently from previous work (e.g., [29]) that perform a simple summation of the individual nodes to measure end-to-end latency, our measurement is more trustworthy since contention and communication overheads are also assessed. Furthermore, we analyze a broader number of computation paths, hence increasing the identification of performance bottlenecks. To measure the end-to-end latency we trace the timestamp of messages since they first entered the system until they are published by the final perception nodes. We track initial inputs through different nodes by tracking down the header information of the messages, which is passed along the subscribe-publish mechanism in ROS. Note that we profile the entire Autoware software stack, with multiple ROS nodes running in parallel. Therefore, unlike previous work [8], we consider interference among nodes, contention, and communication overheads. We show this is key for accurate estimation of latency and performance variability.

System-wide utilization and power dissipation. Complementary to the latency assessment we also gather overall information regarding CPU and GPU utilization. We measure the timeshare different nodes take in both platforms and also how their power dissipation varies. The goal is to find resource-hungry nodes, so we can identify nodes that consume most of the energy. We can also identify if some specific node may require a dedicated platform to run on, or if their execution affects the performance of other nodes. We use the *atop* tool to measure the CPU utilization, collecting utilization every second (the finest grain possible on the tool), while using *nvidia-smi* to obtain GPU utilization and power dissipation, also every second.

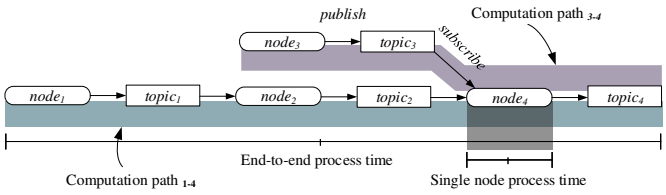


Fig. 4: The latency measurements accounted in our experimentation. Single node process time corresponds to the processing inside a node, excluding communication overheads. End-to-end process time measures the worst case latency since a frame (e.g., camera image, lidar point-cloud) enters the system until its final contribution to the AVs' perception.

Architecture-level characterization. In the final step of our characterization, we rely on hardware counters to obtain further details of important nodes (inferred by the previous two steps), collecting several metrics such as cache accesses and misses, Instructions per Cycle (IPC), cycle counts, instruction mix, branch misprediction rate, the share of time in the CPU/GPU (for nodes that run on both platforms) and the CPU/GPU memory bandwidth. The analysis of this data is extensively covered in our Characterization Analysis, Section IV. To collect the architecture-level metrics we use the *PAPI* library and *valgrind*, for CPU, and the *nvprof* tool for the GPU. This data will serve to support insights and claims during the analysis.

C. Methodology Acknowledgements

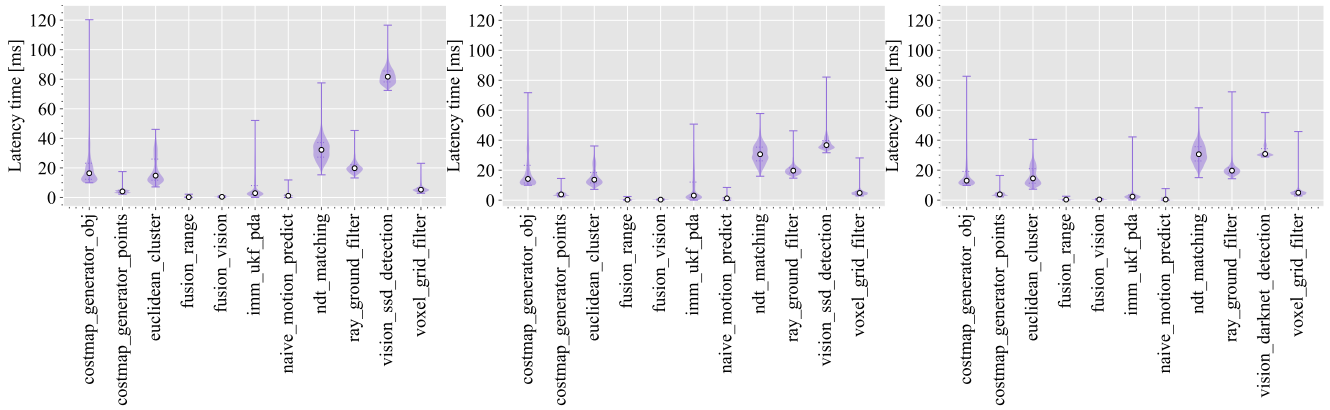
As aforementioned, the lack of an HD map with rich annotation imposes a few limitations. Since we do not have the annotation for traffic light poles position, we cannot perform traffic light detection algorithms. Also, since there is no specification of roads, allowed lanes, and speed limits, we cannot stimulate control and motion algorithms, namely *op_planning, pure_pursuit, and twist_filter*. It is possible to use game-engine based simulators [30], which provide this information in a simulated traffic environment. However, such simulators demand high CPU/GPU utilization. Given the considerable overheads introduced to the system, these simulators are more suited to verify functionalities instead of profiling. Finally, the profiling focuses on perception nodes, as they represent the vast majority of the execution time [8], [29].

IV. Characterization Analysis

This section presents the characterization of Autoware. We first provide an analysis of Autoware's latency, followed by the system-wide utilization and energy consumption, and finally entering in more detail about relevant nodes.

A. Latency Characterization

Single node latency. Figure 5 presents the latency distribution (y-axis) for different nodes, when using different algorithms for image detection. Each series in the plot depicts a node's range of latency values (y-axis), and also how the latency values are distributed (indicated by the shaded thickness). White circles identify the mean values for each series, and two dashed lines limit the first and third quartile of the data distribution, whereas the two continuous lines show the minimum and maximum latencies. The computation path each node belongs to (see Figure 4) will be detailed later.



(a) Autoware with SSD512 image detector. (b) Autoware with SSD300 image detector. (c) Autoware with Yolo image detector.

Fig. 5: Autoware's single node latency for different nodes. Experimentation include different image detector nodes.

We start discussing the mean values, as they suggest the typical latency values, to identify critical nodes on the Autoware software-stack. In all three scenarios *ndt_matching*, *ray_ground_filter*, and *vision_detection* have average latency above 20ms, being the *vision_detection* the more critical, regardless the vision detection algorithm. For instance, if Autoware runs with SSD512 (Figure 5a), the vision detection algorithm takes more than 80ms in its mean latency. If we consider a 100ms as the target time constraint for reaction time, as in previous works [8], [27], we are devoting at least 80% of the time only to process the image detection task. When we consider other options such as SSD300 (Figure 5b) and YOLO (Figure 5c), average latency improves, being under 40ms. Although latency plays a major role in choosing the image detection solution for AVs, we acknowledge that assessing the most propitious image detector is out of the scope of this work, since other metrics which are not evaluated here (e.g. detection precision) also need to be taken into account.

We enrich the analysis considering the tail latency of the nodes. Although values are mostly around the mean, where the shaded thickness is more apparent in Figure 5, the latency of the nodes can occasionally increase and create *latency outliers* during the execution. In the case of an AV, these outliers cannot be neglected since they can cause the system to miss a time deadline for a frame, which in turn introduce a safety flaw. Observing the tail latency serves to identify a broader number of nodes with performance threats for the system. For instance, *costmap_generator_obj*, *euclidean_cluster*, and *imm_ukf_pda_tracker* reach peak latency above 20ms in all three scenarios. Sometimes their latency can scale up by a great margin, as *costmap_generator_obj* taking up to 120ms when running alongside with *SSD512* (Figure 5a). Besides the intrinsic time variability from CPU scheduling, these nodes also take different amounts of time to perform depending on the number of objects in the driving scene. For instance, the more the driving players, the higher the time to track each of them (*imm_ukf_pda_tracker*), project their occupancy site in the world (*costmap_generator_obj*), and obtain their cluster centroids and distance (*euclidean_cluster*).

TABLE III: Dropped messages during Autoware execution

	Topic	Subscribed by Node	Dropped Messages
With SSD512	/detection/fusion_tools/objects	*imm_ukf_pda_tracker*	0.1%
	/image_raw	*SSD512*	16.3%
	prediction/motion_predictor/objects	*costmap_generator_obj*	0.6%
With SSD300	/detection/fusion_tools/objects	*imm_ukf_pda_tracker*	0.1%
	/image_raw	*SSD300*	0.0%
	prediction/motion_predictor/objects	*costmap_generator_obj*	1.0%
With Yolo	/detection/fusion_tools/objects	*imm_ukf_pda_tracker*	0.2%
	/image_raw	*YOLO*	0.0%
	prediction/motion_predictor/objects	*costmap_generator_obj*	1.0%

Figure 5 also shows another interesting finding: the tail latency of some relevant nodes is highly affected by the other components running in parallel. For example, the *costmap_generator_obj* shows a tail latency of 72 ms when using SSD300 as the image detector, but it increases to 120 ms when using SSD512. This large increase of more than 66% in tail latency is due to the interactions among different nodes, as they run in parallel and compete for shared resources. Other nodes such as *ndt_matching* and *voxel_grid_filter* also see a significant increase in tail latency of 34% and 97% respectively depending on the image detection module employed.

Finding 1: *Profiling individual components in isolation leads to a large underestimation of tail latency. The entire system must be profiled to account for communication costs and contention among nodes.*

To complement the latency analysis, let us examine the messages exchanged in Autoware, depicted in Table III. The table contains topics that presented at least one dropped message, together with the dropping percentage. Messages are dropped from a topic every time a newer message arrives on that topic *and* the previous message (which is dropped) was not yet consumed by the destination node[2]. This typically indicates the node is struggling to compute its inputs in time, which can occasionally occur if the single node latency increases.

We underscore the high amount of dropped messages in *SSD512*, whose long processing time results in more

than 16% of messages being dropped. Other nodes such as *imm_ukf_pda_tracker* and *costmap_generator_obj* also present some degree of message dropping. Hence, each node must always complete execution before a new input arrives, assuring messages are always processed. In general, the maximum single node latency is ultimately defined by the input sensor frequency (e.g. LiDAR or camera), which defines the minimum processing pace all nodes should be able to run. Otherwise, if a single node takes too long to execute, there will be message dropping and the whole input processing will not complete, leaving the AV with no perception information for that frame.

Moreover, whereas any message drop is a potential security flaw, the small percentage of message losses for *imm_ukf_pda_tracker* and *costmap_generator_obj* nodes (1% or less message drops) indicates that latency values rarely reach the peak latency depicted in Figure 5. This accentuates the importance of stimulating the AV system on a varied number of situations to capture such flaws.

End-to-end latency. The autonomous-driving task depends on the collaboration of multiple nodes, as introduced in Section II. We now examine different computation paths that Autoware relies on to perceive the environment. We consider the latency of four computation paths, which are presented in Figure 6. The set of nodes and topics that compose each computation path are described in Table IV.

As depicted in Figure 6, the perception latency for different computation paths reaches the order of hundreds of milliseconds. Further, regardless of the chosen image detector algorithm, our measurements show that the end-to-end latency (worst case among all computation paths) always reaches more than 200 ms to compute when considering tail latency. Thereby, there are moments during the AV driving where the system takes more than twice the time commonly stipulated to drive safely (100 ms). This is an interesting finding considering our measurements are over a mature self-driving project. Notwithstanding, we highlight our computing platform (see Table II) is a high-end computer, and larger computing arrangements are hardly an option for AVs due to size and energy consumption constraints. This indicates self-driving algorithms and computing platforms must be improved even further to assure the required performance.

Finding 2: *The Autoware software solution in combination with a high-end computing platform is not able to assure end-to-end perception latency under 100 ms during a complete driving task.*

The more demanding computation paths are those that extract semantics (with object detection or clustering), and perform tracking and prediction. If *SSD512* is used (Figure 6a), then the path with the vision detection (*costmap_vision_obj_ssd*) holds the worst average latency. When faster vision detection algorithms are used, e.g., *SSD300* (Figure 6b) or *YOLO* (Figure 6c), the worst average latency relies on the computing path with the *euclidean_cluster*, namely *costmap_cluster_obj*. Revisiting the single node latency data (Figure 5) together with the end-to-end path analysis, we conclude that *vision_detection*, *ray_ground_filter*, and *euclidean_cluster* are the most important nodes where optimization efforts should focus, since they are time-consuming while also being part of the critical end-to-end path.

[2]ROS topics are queues. Messages are dropped only when a new message arrives and finds the queue full. In the Autoware implementation, however, queues have the size of one. With this, Autoware prioritizes newer frames by keeping only the most up-to-date message in the queue, and avoids investing time to process old messages that no longer describe the current environment.

B. System-Wide Utilization and Power Dissipation

Following the latency analysis, we present system-wide measurements. This guides the understanding of how Autoware utilizes the available computation resources.

Computing platforms utilization. Table V presents the mean time share each Autoware perception node utilized during our experimentation. Notice that only *vision_detection* and *euclidean_cluster* have a share of execution in the GPU.

At first sight, we can observe that changing the *vision_detection* algorithm has some impact on the CPU utilization, going from the most CPU-hungry node when running *SSD512*, but the second most when running with *SSD300* or *YOLO*. Particularly, *YOLO* uses less than half of the CPU compared to *SSD512*. Even more considerable is the impact on GPU utilization, where, for instance, values drop from 26% with *SSD512* to around 16% when using *SSD300*.

The behavior of the remaining nodes is not profoundly affected by the different *vision_detection* algorithms experimented, but rather have their utilization share and ordering (most CPU-hungry to least CPU-hungry) kept along with the different scenarios. We highlight the GPU share for *euclidean_cluster* drops considerably when running alongside with *SSD300*. This accompanies the fact that *SSD300* is the least resource-consuming implementation on the GPU, which can relieve GPU stress improving algorithms running alongside in the GPU, such as the *euclidean_cluster*.

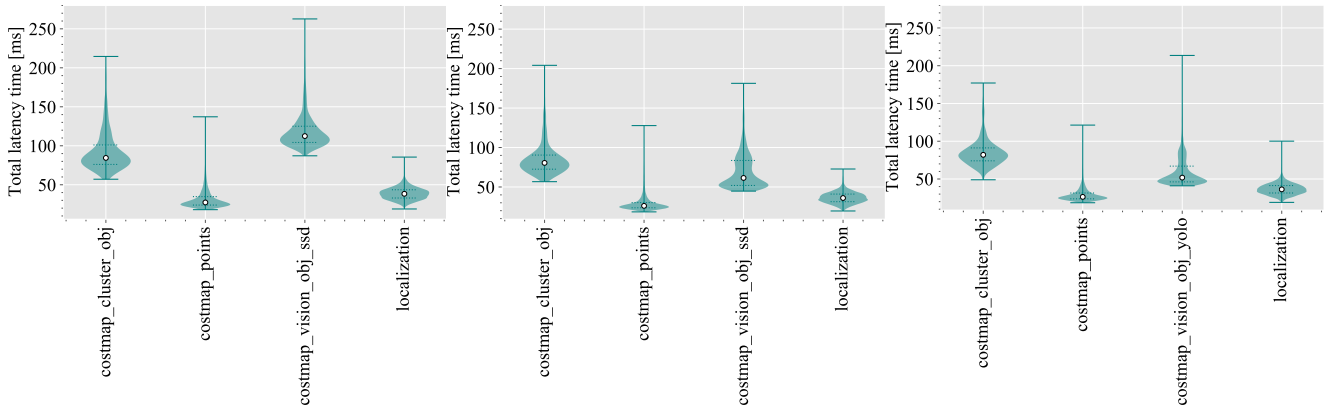
When verifying the total utilization it is possible to see that the utilized setup is not under pressure, with CPU and GPU utilization under 40% in average.

Finding 3: *Resource availability is not a limiting factor for the software-stack and higher performance should be achieved with a more efficient implementation.*

Power dissipation. Complementary to the resource utilization, we also assess the mean power dissipation during Autoware execution, for both the CPU and GPU. Table VI presents this data. We observe that CPU mean power values are generally smaller and vary less with the vision detection algorithm than GPU power dissipation. Since all nodes utilize the CPU, improving or changing a single node (as we change the vision detection algorithm) does not induce great impacts to reduce the total power consumption. Also, not only the nodes but also the complete Operating System and ROS stack are running on the CPU, which diminishes the overall contribution from each individual node.

TABLE IV: Computation paths description

Computation Path	Path description (/topic → node)
localization	/points_raw → voxel_grid_filter → /filtered_points → ndt_matching
costmap_points	/points_raw → ray_ground_filter → /points_no_ground → costmap_generator
costmap_vision_obj_*	/image_raw → vision_ssd_detection → /detection/image_detector/objects → range_fusion_01 → /detection/fusion_tools/objects → imm_ukf_pda_01 → /detection/object_tracker/objects → ukf_track_relay → /detection/objects → naive_motion_predict → /prediction/motion_predictor/objects → costmap_generator
costmap_cluster_obj	/points_raw → ray_ground_filter → /points_no_ground → lidar_euclidian_cluster_detect → /detection/lidar_detector/objects → range_fusion_01 → /detection/fusion_tools/objects → imm_ukf_pda_01 → /detection/object_tracker/objects → ukf_track_relay → /detection/objects → naive_motion_predict → /prediction/motion_predictor/objects → costmap_generator



(a) Autoware with SSD512 image detector. (b) Autoware with SSD300 image detector. (c) Autoware with Yolo image detector.

Fig. 6: Autoware's end-to-end perception latency. Experimentation considers the usage of different image detector nodes.

TABLE V: CPU and GPU utilization share among Autoware nodes

Node	CPU Usage %			GPU Usage %		
	With SSD512	With SSD300	With YOLO	With SSD512	With SSD300	With YOLO
vision_detection	**12.95%**	7.41%	6.51%	**26.00%**	**15.98%**	**30.47%**
costmap_generator_obj	8.60%	**8.23%**	**8.51%**			
ndt_matching	4.15%	4.01%	4.03%			
euclidean_cluster	3.71%	3.37%	3.44%	5.19%	3.19%	5.39%
imm_ukf_pda_tracker	2.84%	3.00%	2.81%			
ray_ground_filter	2.79%	2.80%	2.42%			
range_vision_fusion	1.57%	1.61%	1.64%			
voxel_grid_filter	1.13%	1.05%	1.08%			
naive_motion_prediction	0.71%	0.70%	0.66%			
Total	38.44%	32.18%	31.11%	31.18%	19.17%	35.86%

TABLE VI: CPU and GPU mean power dissipation

	CPU	GPU	Total
With SSD512	44.90 W	122.14 W	167.05 W
With SSD300	42.63 W	67.08 W	109.71 W
With YOLO	42.35 W	116.73 W	159.08 W

In the GPU side, otherwise, improving or changing a single algorithm may be much more impactful. For instance, moving from *SSD512* to *SSD300* reduces 34% of the total power dissipation. With the previous observations, we can argue that nodes using GPU are preferred candidates for improvement if reducing power is the main goal.

C. Architecture-Level Characterization

At this point, we have a broad high-level understanding of Autoware regarding latency, resource utilization, and power dissipation. Based on the aforementioned analysis we have selected a subset of critical nodes, to examine them in depth.

Core microarchitecture metrics. Table VII details a list of *six* nodes from Autoware software-stack, and a data collection based on hardware-counters (for the CPU execution). The nodes were selected for their distinguished impact on the overall system, having either meaningful processing latency or CPU/GPU utilization. Additionally, we present two figures: Figure 7 presents the instruction mix for the evaluated nodes; and Figure 8, which gives further details on the latency distribution of the two vision detection algorithms evaluated in this section (*SSD512*, and *YOLO*).

We start our microarchitecture analysis by considering the two vision detection algorithms (*SSD512*, and *YOLO*). As shown in Figure 8, both nodes largely differ in their CPU usage. The *SSD512* spends more than half of its processing time running on the CPU, whereas *YOLO* spends more than 90% of the time in the GPU. This means that CPU hardware-counters (presented in Table VII) are more significant to *SSD512*. With that in mind, we see that both *SSD512* and *YOLO* have moderate to low IPC. For *SSD512*, which holds the worst IPC among all nodes, this is a consequence of the large branch misprediction rate (9.78%). When analyzing further, we discovered that 71% of CPU time of *SSD512* was

executing a sorting algorithm in the output layer of its Convolutional Neural Network (CNN), utilized by *SSD512* to rank the predicted rectangle boxes that delimit detected objects. Because the branches inside the sorting will depend on the unpredictable input (data to be sorted), the CPU struggles to reach higher accuracy branch predictions. Both vision detection algorithms also present a decent memory locality when running in the CPU, given the moderate L1 cache misses for reads, and especially for writes.

Following, we examine *euclidean_cluster*. We can see from Figure 7 that 50% of its CPU instructions are loads/stores. Also, we derive from Table VII that *euclidean_cluster* has the highest L1 miss rate for reads (4.66%) and writes (5.21%). The *euclidean_cluster* operates on point-cloud data, whose irregular structure imposes poor memory locality. Therefore, a deeper study on the memory access pattern for this node may expose fruitful paths to achieve a better implementation.

Another point-cloud based node, *ndt_matching*, presents a different behavior. Although loads and stores are also dominant (sum up to 52% of the executed instructions), memory locality (in terms of L1 misses) is better when compared to *euclidean_cluster*. We recall, however, that *ndt_matching* receives a down-sampled point-cloud from *voxel_grid_filter* (see Table IV), which may alleviate the memory concerns for this node. We also investigated why the loads and stores account for so much in the instruction mix and discovered that in more than 90% of its CPU time, *ndt_matching* runs code from the point-cloud library [31]. For the most part, it is manipulating tree-like data structures that contain the collections of points the algorithm is trying to match at the moment. Nevertheless, traversing those data-structures can also explain the considerable branch misprediction faced by the node, which sums up to more than 3%.

We then analyze the *imm_ukf_pda_tracker* node. First, its instruction mix indicates a more control-flow behavior. Even though, branch misprediction is low and locality does not appear as a threat since L1 miss rate is low.

Finally, we investigate the *costmap_generator_obj* node. As we can see from the instruction mix (Figure 7), this node is more computation-bound, with the fewest share of load/store instructions among all nodes. Because of this, *costmap_generator_obj* presents a good IPC of 2, and low values for both L1 misses and branch mispredictions. Overall, this node has propitious characteristics to execute in the CPU.

Beyond single node analysis. Our work profiles a complete AV environment, with multiple nodes running at the same time and collaborating to achieve the driving-task successfully. Differently from previous works, which often analyze nodes individually, our approach leads to a more realistic characterization since the different nodes executing may impact on one another. Following, we present a quantitative comparison of both approaches: running a node alone or running a node together with its counterparts.

We perform this experiment on two image detection nodes from Autoware, namely *SSD512* and *YOLO*. We present their mean execution time and standard deviation in Figure 8.

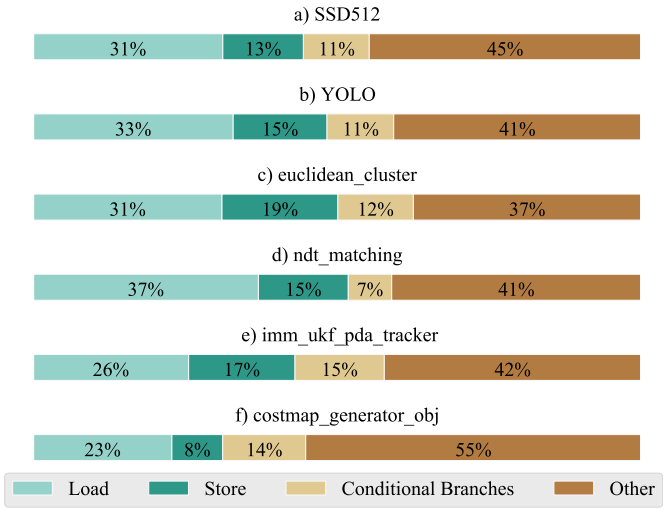


Fig. 7: The instruction mix for the Autoware nodes.

Two main points arise from these experiments. First, when nodes share the computing platform, running alongside other nodes, their mean latency increase. For instance, *SSD512* has a mean latency of 73.45 ms, when executing alone, but the mean latency increases up to 82.26ms when all nodes in Autoware are also performing; an increase of 12%. As for the *YOLO* node, the mean latency rises from 31.23 ms, when running alone, to 33.14 ms, when the other nodes are also executing; a 6% increase in the mean latency.

Finding 4: *Using single node measurements to assess latency bottlenecks can be a pitfall, since threats may be under-considered as nodes perform better, in the average, when running apart from the full system.*

The second interesting finding is the impact on the standard deviation. As we can assess from the error bars, sharing the computing platform among multiple nodes increases the standard deviation of the node's latency. For *SSD512* the standard deviation goes from 1.01 ms, running alone, to 4.81 ms, when the complete set of nodes is running. Similarly, the standard deviation for *YOLO* node latency goes from 0.88 ms, running the node alone, up to 4.05 ms, when we execute all nodes alongside.

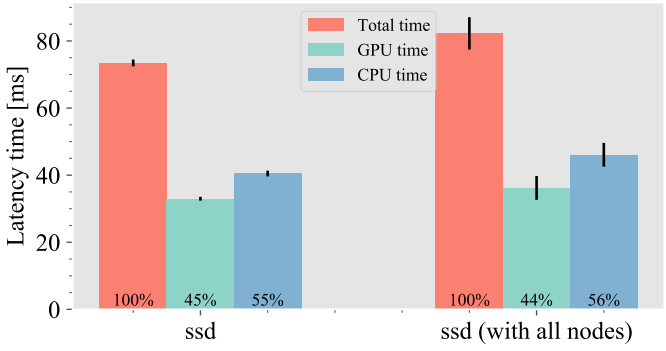
Finding 5: *When all nodes are executing, the predictability of the latency for each node is weakened. Since AVs' should cope with worst-case scenarios, such as the ones who cause tail latency, full system experimentation is best suited for profiling purposes. In this case, the node's variability increases, causing unpredictable behavior that cannot be found when nodes run alone.*

V. Related Work

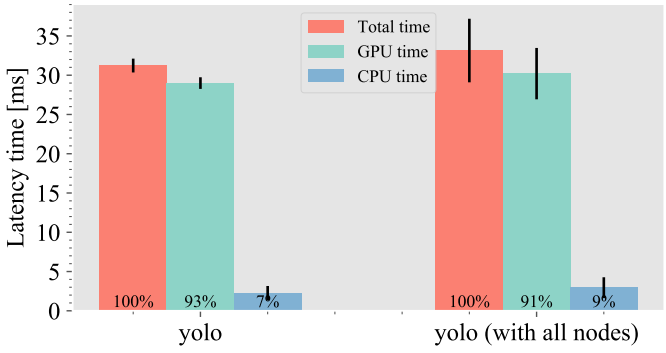
Autonomous Vehicles (AVs) have attracted the attention of industry [32], [33] and academia [11] in recent years. Prior work [8], [29], [34] focused on characterizing the performance of computing systems for AVs and understanding the main bottlenecks. In this section, we review some of these previous works and explain the differences with our characterization.

TABLE VII: Microarchitecture Profiling of Different Autoware Nodes

	SSD512	*YOLO*	*euclidean_cluster*	*ndt_matching*	*imm_ukf_pda_tracker*	*costmap_generator_obj*
Instructions per Cycle	1.03	1.36	1.36	1.26	1.14	2.07
L1 miss rate (read)	2.36%	3.88%	4.66%	1.37%	1.55%	0.20%
L1 miss rate (write)	0.95%	0.45%	5.21%	0.52%	1.51%	0.29%
Branch misprediction	9.78%	0.10%	1.20%	3.06%	0.76%	0.11%



(a) SSD512 image detector.



(b) Yolo image detector.

Fig. 8: The CPU and GPU participation on the total latency for SSD512 and Yolo image detectors. Two measurements were performed: vision detection nodes running standalone, and alongside all other nodes.

Lin et al. [8] analyze the requirements of autonomous driving systems and quantify the latency and power of different components on CPUs, GPUs, FPGAs and ASICs. Our work is different in several ways. First, they only consider camera-based AVs, whereas we characterize a system that includes both cameras and LiDAR, that we believe is more representative of industry-level AVs such as Waymo's self-driving car [35]. Note that LiDAR related components, such as euclidean clustering, exhibit significant tail latency and may prevent real-time performance (see Section IV). Second, they profile isolated components, whereas we consider the full AV system to capture communication time, interference among different nodes and contention for shared resources. We report a large increase in latency variability when running the full system and, therefore, we claim that the entire system must be considered to obtain representative results.

Kato et al. [29] analyze the performance of Autoware on an NVIDIA DRIVE PX2. They run the entire software stack and measure end-to-end latency. However, they only consider the latency of a single path from sensor to actuation that includes localization and motion planning, but excludes some of the main bottlenecks such as image detection. Furthermore, they ignore the communication costs associated with publish/subscribe mechanism in ROS. Our approach is different as we profile a much boarder number of paths and we report end-to-end latency as the longest path, including the execution time of all the relevant nodes and the communication time.

Toschi et al. [34] present a characterization of the perception modules of Apollo, Baidu's self-driving car solution. They report execution time of individual components, whereas we analyze end-to-end latency and power dissipation, and we use a full open source solution, Autoware.

VI. Conclusions

This paper presents a detailed characterization of Autoware, a modern autonomous driving system, based on a robust methodology that profiles the full software-stack end-to-end. This approach is novel in several ways. First, it includes contention among multiple nodes that run in parallel and communication costs. Our results show that these factors severely impact both tail latency and performance predictability, largely differing from the measurements obtained when profiling isolated nodes. Second, we consider both image processing nodes and LiDAR related components. Finally, we characterize latency and power of the critical end-to-end path, show that a modern hardware platform cannot guarantee real-time performance, and identify candidates for software optimization and hardware acceleration.

VII. Acknowledgments

This work has been supported by the the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057) and the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU).

References

[1] C. Urmson and W. Whittaker, "Self-driving cars and the Urban challenge," *IEEE Intelligent Systems*, vol. 23, no. 2, pp. 66–68, 2008.
[2] US National Highway Traffic Safety Administration, "Automated Driving Systems 2.0: A Vision for Safety," Tech. Rep., 2017.
[3] "Waymo." [Online]. Available: https://waymo.com/
[4] Nvidia, "NVIDIA DRIVE Overview," 2019. [Online]. Available: https://www.nvidia.com/en-us/self-driving-cars/
[5] Tesla, "Autopilot — Tesla," 2020. [Online]. Available: https://www.tesla.com/autopilot
[6] Baidu, "Apollo." [Online]. Available: http://apollo.auto/

[7] Y. Wang, "Putting the brain in driverless vehicles with MDC," *Huawei Communicate Issue 86*, no. 86, pp. 42–45, 2019. [Online]. Available: https://www.huawei.com/uk/about-huawei/publications/communicate/86/driverless-vehicles-with-mdc

[8] S. C. Lin, Y. Zhang, C. H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 751–766, 2018.

[9] A. Foundation, "The Autoware Foundation - Open Source for Autonomous Driving." [Online]. Available: https://www.autoware.org/

[10] Autoware, "Autoware.AI · GitLab." [Online]. Available: https://gitlab.com/autowarefoundation/autoware.ai

[11] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[12] Baidu, "ApolloAuto/apollo: An open autonomous driving platform." [Online]. Available: https://github.com/ApolloAuto/apollo

[13] Autoware, "Autoware Wiki." [Online]. Available: https://gitlab.com/autowarefoundation/autoware.ai/autoware/-/wikis/home

[14] H. G. Seif and X. Hu, "Autonomous Driving in the iCity—HD Maps as a Key Challenge of the Automotive Industry," *Engineering*, vol. 2, no. 2, pp. 159–162, 2016. [Online]. Available: http://dx.doi.org/10.1016/J.ENG.2016.02.010

[15] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," Tech. Rep. [Online]. Available: http://stair.stanford.edu

[16] P. Biber and W. Strasser, "The normal distributions transform: a new approach to laser scan matching," in *International Conference on Intelligent Robots and Systems*, vol. 3. IEEE, 2003, pp. 2743–2748. [Online]. Available: http://ieeexplore.ieee.org/document/1249285/

[17] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9905 LNCS. Springer Verlag, 2016, pp. 21–37.

[18] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

[19] M. Sualeh and G. W. Kim, "Dynamic Multi-LiDAR based multiple object detection and tracking," *Sensors (Switzerland)*, vol. 19, no. 6, 2019.

[20] M. Schreier, V. Willert, and J. Adamy, "Compact Representation of Dynamic Driving Environments for ADAS by Parametric Free Space and Dynamic Object Maps," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 2, pp. 367–384, 2016.

[21] A. S. A. Rachman, "3D-LIDAR Multi Object Tracking for Autonomous Driving Multi-target Detection and Tracking under Urban Road Uncertainties," Ph.D. dissertation, 2017.

[22] Y. Ninomiya, N. Akai, H. Darweesh, E. Takeuchi, K. Takeda, A. Sujiwo, Y. Morales, T. Tomizawa, and S. Kato, "Open Source Integrated Planner for Autonomous Navigation in Highly Dynamic Environments," *Article in Journal of Robotics and Mechatronics*, 2017. [Online]. Available: https://www.researchgate.net/publication/319201866

[23] R. C. Coulter, "Implementation of the Pure Pursuit Path Tracking Algorithm," *Communication*, no. January, 1992.

[24] T. IV, "Autoware Data." [Online]. Available: https://data.tier4.jp/

[25] "GitHub - weiliu89/caffe at ssd." [Online]. Available: https://github.com/weiliu89/caffe/tree/ssd#models

[26] "YOLO: Real-Time Object Detection." [Online]. Available: https://pjreddie.com/darknet/yolo/

[27] H. Chishiro, K. Suito, T. Ito, S. Maeda, T. Azumi, K. Funaoka, and S. Kato, "Towards heterogeneous computing platforms for autonomous driving," *2019 IEEE International Conference on Embedded Software and Systems, ICESS 2019*, no. December 2018, 2019.

[28] S. Thorpe, D. Fize, and C. Marlot, "Speed of processing in the human visual system," *Nature*, vol. 381, no. 6582, pp. 520–522, 1996.

[29] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems," *Proceedings - 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018*, no. April, pp. 287–296, 2018.

[30] LG Electronics, "LGSVL Simulator — An Autonomous Vehicle Simulator." [Online]. Available: https://www.lgsvlsimulator.com/

[31] "Point Cloud Library — The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing." [Online]. Available: https://pointclouds.org/

[32] D. L. Rosenband, "Inside waymo's self-driving car: My favorite transistors," in *2017 Symposium on VLSI Circuits*, 2017, pp. C20–C22.

[33] M. Dikmen and C. M. Burns, "Autonomous driving in the real world: Experiences with tesla autopilot and summon," in *Proceedings of the 8th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, ser. Automotive'UI 16. New York, NY, USA: Association for Computing Machinery, 2016, p. 225–228. [Online]. Available: https://doi.org/10.1145/3003715.3005465

[34] A. Toschi, M. Sanic, J. Leng, Q. Chen, C. Wang, and M. Guo, "Characterizing perception module performance and robustness in production-scale autonomous driving system," in *Network and Parallel Computing*, X. Tang, Q. Chen, P. Bose, W. Zheng, and J.-L. Gaudiot, Eds. Cham: Springer International Publishing, 2019, pp. 235–247.

[35] Waymo, "Introducing the 5th-generation Waymo Driver." [Online]. Available: https://blog.waymo.com/2020/03/introducing-5th-generation-waymo-driver.html